Information and Software Technology 56 (2014) 749–762

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Model-based testing of global properties on large-scale distributed systems

Gerson Sunyé [a,*], Eduardo Cunha de Almeida [b,1], Yves Le Traon [c,1], Benoit Baudry [d], Jean-Marc Jézéquel [d]

[a] Lina - Université de Nantes, 2 rue de la Houssinière, BP 92208 44322, Nantes Cedex 03, France
[b] Departamento de Informática, Rua Cel. Francisco H. dos Santos, 100 Centro Politécnico, Jardim das Américas - Curitiba - PR, Caixa Postal: 19081, CEP 81531-980, Brazil
[c] Faculty of Science, Technology and Communication, 6, rue Coudenhove-Kalergi, L-1359 Luxembourg-Kirchberg, Luxembourg
[d] IRISA Rennes, Campus universitaire de Beaulieu, 263 Avenue du Général Leclerc - CS 74205, 35042 RENNES Cedex, France

ARTICLE INFO

Article history:
Received 22 August 2013
Received in revised form 4 February 2014
Accepted 5 February 2014
Available online 14 February 2014

Keywords:
Software testing
Distributed software
Model-based testing

ABSTRACT

Context: Large-scale distributed systems are becoming commonplace with the large popularity of peer-to-peer and cloud computing. The increasing importance of these systems contrasts with the lack of integrated solutions to build trustworthy software. A key concern of any large-scale distributed system is the validation of global properties, which cannot be evaluated on a single node. Thus, it is necessary to gather data from distributed nodes and to aggregate these data into a global view. This turns out to be very challenging because of the system's dynamism that imposes very frequent changes in local values that affect global properties. This implies that the global view has to be frequently updated to ensure an accurate validation of global properties.
Objective: In this paper, we present a model-based approach to define a dynamic oracle for checking global properties. Our objective is to abstract relevant aspects of such systems into models. These models are updated at runtime, by monitoring the corresponding distributed system.
Method: We conduce real-scale experimental validation to evaluate the ability of our approach to check global properties. In this validation, we apply our approach to test two open-source implementations of distributed hash tables. The experiments are deployed on two clusters of 32 nodes.
Results: The experiments reveal an important defect on one implementation and show clear performance differences between the two implementations. The defect would not be detected without a global view of the system.
Conclusion: Testing global properties on distributed software consists of gathering data from different nodes and building a global view of the system, where properties are validated. This process requires a distributed test architecture and tools for representing and validating global properties. Model-based techniques are an expressive mean for building oracles that validate global properties on distributed systems.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Large-scale distributed systems are becoming commonplace with the increasing popularity of peer-to-peer (P2P) or cloud computing. For instance, the Gnutella [1] P2P system shares petabytes of data among millions of users. Data intensive applications, based on Google's MapReduce [2], process several petabytes of data every day, on large clusters of commodity machines, in a way that is also resilient to machine failures.

The high popularity of these systems contrasts with the lack of integrated test solutions to ensure their general quality under normal and abnormal conditions. A main reason is the complexity of reproducing a real world environment together with a non-intrusive test environment. This is because the scale of the system affects several test components, such as: test controllability [3], fault-injection [4], logging facilities, and oracle calculation, among others. In the precise case of the oracle, the validation of global properties becomes a major problem, as they depend on values that are spread throughout the system.

This problem is faced when testing, for instance, the global correctness of the routing algorithm of a P2P system. In these

* Corresponding author. Tel.: +33 299847291.
E-mail addresses: gerson.sunye@univ-nantes.fr (G. Sunyé), eduardo@inf.ufpr.br (E.C. de Almeida), yves.letraon@uni.lu (Y. Le Traon), bbaudry@irisa.fr (B. Baudry), jezequel@irisa.fr (J.-M. Jézéquel).

[1] Supported by the Fonds National de la Recherche, Luxembourg, TOOM Project Grant: C12/IS/4011170.

http://dx.doi.org/10.1016/j.infsof.2014.02.002
0950-5849/© 2014 Elsevier B.V. All rights reserved.

systems, efficient message routing depends on the correct state of local routing tables, which must be maintained frequently, according to the dynamic state of the system: arbitrary network latencies, node failures, and churn. Hence, the actual content of a routing table is nondeterministic and highly dynamic, making it non-obvious to tell whether it is correct at any given point in time. The global correctness of the routing algorithm depends on the (volatile) content of the routing tables in each local node that is very hard to aggregate into a global view in a timely and scalable way. Furthermore, the correctness of the global view can only be verified at given states: it may be invalid right after churn, but must be valid after a certain delay in a stable state.

A typical approach for testing such a feature in a distributed system consists of a centralized controller and several testers, each one controlling a single port or node interface [5]. The tester is the application that runs in the same logical devices as system nodes, and controls their execution and their volatility, making them leave and join the system at any time, according to the needs of a test. The controller sends the test inputs, controls the synchronization of the test case execution and receives the outputs (or local verdicts) from each tester [6]. However, building a global verdict from the information gathered locally can be a very difficult problem. For instance, in a system where each node maintains a set of references to its physically closest neighbors (e.g., Pastry [7]), the only way to validate the correct construction of the system would be to first gather information from all nodes, then calculate the distance between them, and finally check if the contents of the reference set are actually the closest neighbors. A similar problem arises when verifying load balance on MapReduce systems, which distribute their load burden across their nodes, including storage, query processing, and computations. To verify their algorithm of load-balancing, one must gather information from all nodes, which can be a large amount of data, and check for system usage information (e.g., partitioning of datasets).

In this paper, we present an approach leveraging the idea of model at runtime [8] to provide a dynamically built oracle for testing properties in large-scale distributed systems. This approach focuses on global, liveness, observable and controllable properties. More precisely, it focuses on a particular class of properties that cannot be calculated by a single node or by a portion of the system; that are eventually true; that are observable from the system interface; and that respond to external events. We propose to efficiently keep updating a global model of the system during its execution. This model is then instantiated and evolved at runtime, by monitoring the corresponding distributed system, and serve as oracle for distributed tests. On the implementation side, we show that standard Model-Driven Engineering (MDE) technology such as Kermeta [9] can be used to easily implement the oracle part of such model-based distributed tests. We use this approach to test topology-related properties on two open-source, structured P2P systems. This approach extends our previous work [6,10], where we presented a methodology and an architecture to test local properties of large-scale distributed systems, which can be verified locally to each node. The addition of a global model and an efficient update mechanism allows also to test global properties.

The rest of the paper is organized as follows. The next section presents a real world motivation case. Section 3 introduces some fundamental concepts in large-scale distributed systems and some global topology properties these systems must satisfy. Section 4 presents our approach to represent and check these properties, as well as our architecture for testing distributed systems. Section 5, describes our validation through implementation and experimentation on two open-source systems. Section 6 discusses related work. Section 7 concludes.

2. Motivating case: The 2010 Skype outage

On December 22nd 2010, the Skype network suffered a critical failure that lasted approximately 24 h from December 22nd, 16:00 GMT to December 23rd, 16:00 GMT. The failure concerned more than 23,000,000 of online users [11]. Fig. 1 illustrates the outage. When the number of online users was almost reaching its highest point, it suddenly started to drop. In almost 1 h, there were less than 1 million online users.

Skype is a successful example of combing modern distributed architectures to implement a popular, reliable, portable, and

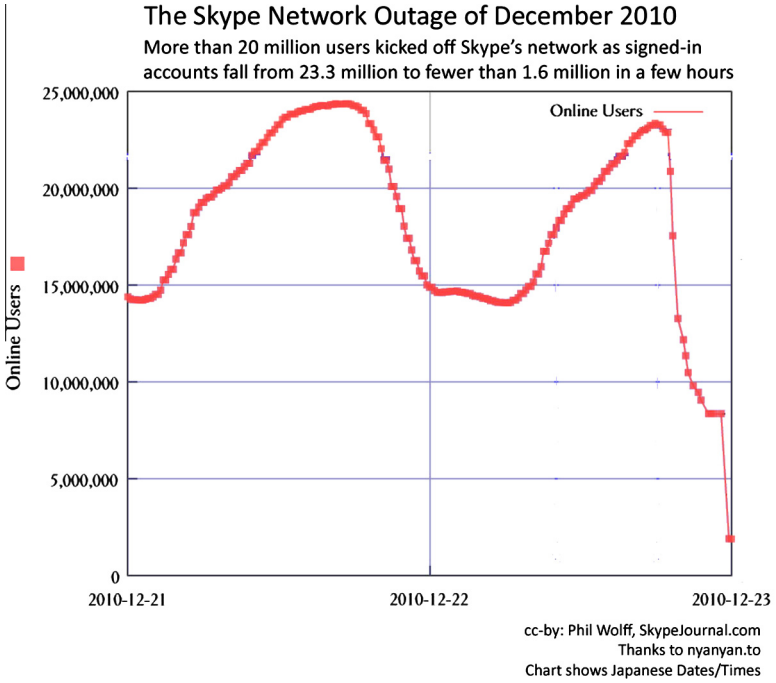

Fig. 1. The Skype network outage. (Images by Phil Wolff. Available under Creative Commons Attribution-Share Alike 2.0 Generic License.)

interoperable software. Indeed, Skype architecture is a harmonic combination of different paradigms, merging centralized, peer-to-peer, and cluster architectures. A centralized login server handles all the network connections and a Distributed Hash Table (DHT) stores user information. Communications between nodes are done through a point-to-point connection, and clusters, which act as a private cloud, provide some services such as group chat or offline messaging.

The outage commenced on December 22nd, when a cluster of support servers responsible for offline instant messaging became overloaded. Because of this overload, some Skype clients received delayed responses from the overloaded servers. Clients using a specific version of Skype for Windows (5.0.0152) did not process properly these delayed responses and crashed.

Users running other versions were not affected by this initial problem. Nevertheless, around 50% of all Skype users globally were running the 5.0.0.152 version of Skype for Windows and the crashes caused approximately 40% of those clients to fail. Among these clients, there were 25% to 30% of the publicly available super-nodes.

Super-nodes are nodes with extra behavior: they help common nodes to join the network and store some user information on the DHT. When users noticed that their clients crashed, they simply relaunched their software. The problem is that super-nodes do not start as super-nodes, they start as common nodes and become super-nodes, if they have enough resources and are stable for a while. As the former super-nodes restarted as common nodes and tried to join the system, some of the remaining super-nodes received a traffic one hundred times greater than normal. Since Skype super-nodes are deployed on client machines, they have a built-in mechanism that avoids having a huge overload in the host machine, halting the super-node when a given threshold is reached. Thus, all super-nodes that reached the threshold left the system, surcharging the remaining super-nodes and driving the whole system into an unavoidable cascade of shutdowns. Fig. 2 illustrates the fall of the super-nodes. From December 22nd at 20:46 until December 23rd 2:16 GMT, 98% of the Skype network super-nodes were offline.

To recover the network, the engineering team added hundreds of new Skype nodes that act as dedicated super-nodes, which should have provided enough capacity to allow the network to bootstrap. However, only a small portion of users (15–20%) were "healing". The team introduced then several thousands of super-nodes, using the resources that support the Group Video Calling. These new super-nodes and the nightfall helped the network to heal. During the night, the full recovery was beginning. On December 23rd at 16:00 GMT, clients could connect normally to the network. When common nodes start becoming super-nodes, engineering could start removing the dedicated ones.

This is the second major Skype outage; the first one dates back to 2007. When analyzing the causes of this outage, we notice two distinct faults:

1. The misinterpretation of server messages that were delayed causing nodes to crash.
2. The incapacity of super-nodes to treat a large number of join requests, which also prevents the system to bootstrap a large number of nodes at the same time.

A conformance testing approach could catch the first fault, if combined with fault injection (to simulate message delays). A unique test driver could individually test endpoints and reproduce the fault. Nevertheless, finding the correct sequence of messages that can drive the node into a faulty state is a complex task. Indeed, the Windows software that crashed was subject to extensive internal testing and months of beta testing with hundreds of thousands of users, without revealing this fault.

The second fault is more complex, because its reproduction is more challenging. A single test driver cannot generate sufficient load to crash a super-node. Here, a different approach is needed, either using several distributed test drivers or reproducing a real-scale scenario. Contrary to the first fault, the sequence of steps that expose this fault is rather simple, either creating a large system instantaneously or disconnecting super-nodes from a stable system. In both cases, a global model of the topology is necessary to identify the nodes that should be disconnected and to verify that the system is sound.

3. Background

Cloud computing offers a highly-scalable environment that runs any kind of system. For instance, large-scale systems with centralized components, such as: GoogleFS (with master nodes), and MapReduce (with job trackers); or decentralized large-scale

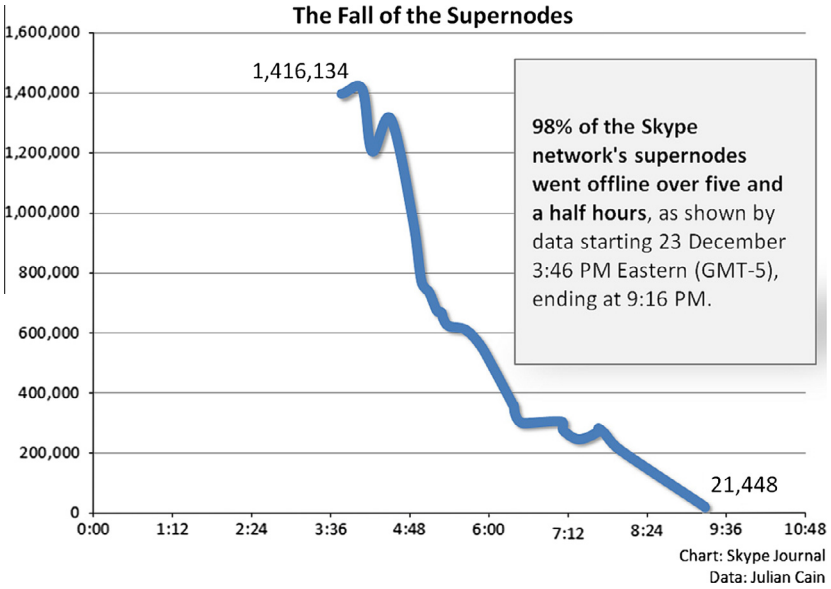

Fig. 2. The fall of the supernodes (Images by Phil Wolff. Available under Creative Commons Attribution-Share Alike 2.0 Generic License.)

systems that fall under the P2P definition, such as Distributed Hash Tables (DHT). The development of these systems is raising new requirements and issues, other than those already tackled during the development of traditional distributed systems:

Large-scale Large-scale distributed systems are expected to connect a large number of nodes (from thousands up to several millions).
Dynamicity Resources may be dynamically added to or removed from the system. This concerns physical nodes, as well as software services.
Heterogeneity of resources The nodes that compose the system are disparate: smartphones, laptops, clusters, supercomputers, etc.
Diversity of purposes These systems are used on different domains, from data sharing to massive processing applications.
Stateless protocols Nodes may receive events defined in their interfaces in any order and at any moment. In essence, communication consist of independent requests-responses pairs.
Symmetry Several nodes play identical roles, ensuring reliability (there is no single point of failure) and load balance (load is distributed symmetrically across nodes).
Elasticity The system scales out and in quickly, adapting itself to load.

3.1. Routing tables

In a large-scale distributed system, nodes have a partial view of the system, i.e., their routing tables keep only a subset of other node addresses. The choice to build the routing table is then crucial for the performance of the whole system. There are as many routing algorithms as there are different systems. In some data sharing systems (e.g., Gnutella, Kazaa), the routing table is built randomly, each node keeps a set of nodes that represent some interest. In structured systems, the routing table is built systematically, each node keeps a set of nodes with the specific ID that are needed for efficient routing.

Currently, there are two major ways to maintain the routing table: actively and lazily. In the former, each node periodically pings all its neighbors and drops the unavailable nodes (e.g., Chord [12]). In the latter, extra status information is added to messages exchanges and the unavailability of a node is only noticed when one of its neighbors does not answer a given query. Some systems (e.g., Bamboo [13], Pastry [7]) use both approaches to maintain their routing tables.

In Pastry, the routing table is divided into three parts. The first one, named *leaf set* contains all nodes having numerically close node ID (i.e., ID that share the same prefix). The second one, the actual *routing table*, contains nodes with different prefixes (at least one node for each element of the prefix domain). The third one, the *neighborhood set*, contains the physically closest nodes, independently from their ID. Pastry uses both, lazy and active approaches to update the routing table. While the leaf and the neighborhood sets are maintained actively, the actual routing table is only updated when a node communicates with its neighbors.

In Chord, each node maintains a routing table with at most m entries, where 2^m is the maximum size of the system. The ith entry in the table at node n contains the identity of the node, s, that succeeds n by at least 2^{i-1} on the identifier circle, i.e., $s = successor(n + 2^{i-1})$ where $1 \leqslant i \leqslant m$. When the ID is not taken, the entry is the node with the following ID. Chord uses an active approach to update its routing table, periodically running a process called "stabilization".

For instance, in a Chord system with $m = 3$ (Fig. 3), containing nodes n_0, n_1, n_3, and n_6, the routing table of n_0 stores the addresses nodes n_1, n_3 and n_6. When a new node n_4 joins the system, then n_0 will update its routing table and replace the address of n_6 with n_4. The routing tables of both systems, Pastry and Chord, are the subject of the experiments presented in Section 5.

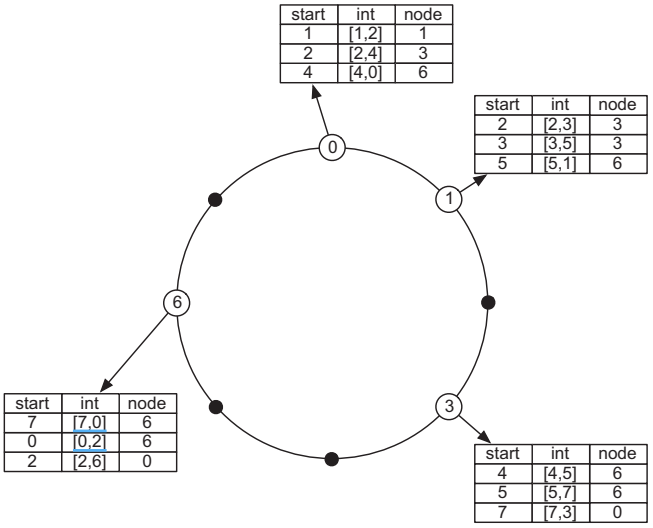

Fig. 3. Chord routing.

3.2. Properties

As any other distributed system, large-scale ones must satisfy the properties of safety and liveness [14]. Safety properties are predicates that should always be true, ensuring that the system never reaches an unacceptable state. For instance, a Chord routing table must never have a null entry: in a system composed of only one node, all entries in the routing table must point to the node itself.

Liveness properties are predicates that should eventually be true. For instance, in Chord, the routing table of a node is valid until another node has joined the system. At this precise state, table entries are and will remain invalid during a *stabilization* time, i.e., the time necessary to detect the presence of a new node and the consequent routing table update. In real life execution, with frequent churn, these properties may never be established.

We list below several topology-related liveness properties, which are tested in Section 5.

Definition 1. Let S be a large-scale distributed system, $Nodes^S$ the nodes composing this system and N the size of the system in terms of number of nodes.

Definition 2. The topology of S can be represented by a directed graph, where each node $n \in Nodes$ is a vertex and every entry in its routing table is an edge to a neighbor.

The first property, which is common to all systems, is the connectivity of the system.

Property 1 (*Connectivity*). *S is a strongly connected graph.*

The second property, which is common to several DHT algorithms [15] (e.g., Pastry, Tapestry, Chord, Kademlia), concerns the diameter of the system, which should be $O(logN)$. It is important to note that this property is too lazy for $O(1)$ DHTs [16] and should be strengthened.

Property 2 (*Diameter*). *The maximum eccentricity over all vertices of S is $O(logN)$.*

While these two first properties are rather simple to verify, if one has a centralized model of the topology, they are essential to test. If they are not respected, the accuracy of the tests presented in Sections 5.4, 5.5 and 5.6, which verify the correct update of the routing table upon churn, would be compromised. Functional tests, such as the correctness of message routing and of data insertion, would not be reliable. The third property concerns the self-organization of DHTs.

Property 3 (*Self-organization*). *When a node joins (or leaves) the system, the total cost to update the routing tables, in terms of the number of messages exchanged, is* $O((\log N)^2)$ *messages.*

The fourth property is the ability to handle churn and remain working properly. Some routing algorithms are liable to fractionate the structure upon churn preventing fractions to send messages to one another. Some solutions are provided to merge fractions [17], but not all the implementations do that.

Property 4 (*Self-healing*). *S remains strongly connected upon churn.*

The properties defined above are related to the topology of peer-to-peer systems. There are two additional properties that are not related to the topology of the system and therefore validation is left for future work, but yet need a global view of the system to be verified: load balance and elasticity. Load balancing is the ability to distribute the workload across the nodes of the system for scalability. For instance, MapReduce, distributed database management systems, and P2P systems distribute their load burden across their nodes. In the particular case of P2P, load balance may rely on consistent hashing algorithms (for structured systems) or on satisfaction load balance algorithms. A possible approach to verify the *Load Balancing* property is to gather information from all the nodes of the system, which can be a large amount of data, and then check the consumption of computing resources.

The elasticity property is the ability of a system to add or remove resources at a fine grain and with a small lead time [18]. Elasticity is ensured either manually or automatically, through the interaction with the infrastructure provider of a cloud system. A possible approach to verify this property is to vary the load of the system and observe the system behavior. While the load varies, we expect the allocation, or decommissioning of failed or surplus nodes.

Unstructured systems, which rely on gossiping protocols are also an interesting class of system for testing global properties, other than the Connectivity presented above. Some examples of these properties are [19]: the efficiency of message propagation, message coverage, message delay, degree distribution (number of neighbors by node), clustering coefficient and the reliability under churn. However, the verification of these properties is part of future work.

3.3. Kermeta

Kermeta is a MDE workbench for building rich development environments around meta-models using an aspect-oriented paradigm [20,21]. It has been designed to easily extend meta-models with many different concerns (such as syntactic correctness including context information, execution information, model transformations, tracing information, and connection to concrete syntax) expressed in heterogeneous languages. A meta-language such as the Meta Object Facility (MOF) standard [22] indeed already supports an object-oriented definition of meta-models in terms of packages, classes, properties, and operation signatures. However, MOF does not include concepts for the definition of constraints or operational semantics (MOF only contains operations signatures). Kermeta can thus be seen as an extension of MOF with a language for specifying constraints and operation bodies at the meta-model level.

The action language of Kermeta is especially designed to process models. It is imperative and includes classical control structures such as blocks, conditional and loops. It implements traditional object-oriented mechanisms for multiple inheritance and behavior redefinition with a late binding semantics. It is statically typed, with generics and provides reflection as well as an exception handling mechanism.

In addition to object-oriented structures, the MOF contains model-specific constructions such as containment and associations between classes. These elements require a specific semantics of the action languages in order to maintain their integrity. For instance, the assignment of a property must handle the other end of the association if the property is part of an association and the object containers if the property is a composition.

Kermeta expressions are a superset of the Object Constraint Language (OCL) ones and have a close syntax. In particular, they include operations similar to OCL iterators on collections such as *each*, *collect*, *select* or *detect*. The standard framework of Kermeta also includes all the operations defined in the OCL standard framework. This alignment between Kermeta and OCL allows OCL constraints to be directly imported and evaluated in Kermeta. Classes define invariants and operations define pre- and post-conditions. The Kermeta virtual machine has a specific execution mode, which monitors these contracts and reports any violation.

3.4. Models at runtime

Models at runtime [23] are formal representations of the system which support computer-based processing, unlike most models commonly used in analysis and design. As stated by Bran Selic:

> This enables formal coupling between models and the systems they represent, similar to the relationship that exists between a program written in a high-level programming language and its machine code counterpart (pp. 26).

In our case, we use models at runtime as a "live" oracle within a test architecture to check properties during the execution of a test sequence. Since we are not able to directly observe the current state of the distributed system, we take snapshots of its nodes periodically, aggregate the information and update the model. Since we focus on liveness properties, we do not need to analyze all states of the system but only given states, after exercising the software interface of one or more nodes.

4. Testing global liveness properties

In this section, we present our approach for testing global liveness properties on large-scale distributed systems. After describing briefly the test architecture, we present our test approach, introduce test cases, explain their implementation, and discuss the limits of the approach.

We consider a test case as a pair $\langle \mathcal{TS}, \mathcal{O} \rangle$, where $\mathcal{TS}$ is a test sequence, i.e., a sequence of steps that drives the System Under Test (SUT) into a given state, and $\mathcal{O}$ is the oracle, which reads the output generated by the SUT and provides a verdict.

4.1. Architecture

The test architecture has two main components, the tester and the controller. The controller executes global test sequences and dispatches test steps through the distributed testers. It uses synchronization messages to ensure that a test step is completely

finished by all testers before starting the next one. Synchronization messages are issued by testers and report the end of execution of a test step, as well as the execution delay and the potential detected execution errors. The test sequence specifies the test execution flow and is deployed on the test controller.

Testers execute test steps and control the volatility of nodes, one tester per node. They only interact with the software interface of the SUT, not with the network. Testers do not interfere in the communication among the nodes of the SUT. The individual control of nodes allows the architecture to simulate complex topology failures, as the failure of Skype super-nodes presented in Section 2, or a similar outage, where node failures caused cascading problems, as it happened to Gmail [24] in February 2009.

The interaction between testers and system nodes is ensured through the use of adapters. Adapters expose the interface of nodes, defining which test steps can be remotely called in the global test case, similarly to TTCN-3 adapters [25]. As in TTCN-3-based tools, the adaptation cannot be generated automatically and must be hard-coded.

The UML deployment diagram presented in Fig. 4 illustrates the deployment of the framework: each tester runs on a logical node (the same as the node it controls) and is connected to a test controller. Testers are independent from the node interface. The interaction between testers and nodes is ensured by adapters, which are loaded dynamically by testers during deployment.

To ensure the scalability of the framework when testing large-scale systems, testers are organized in a tree topology (Fig. 5). In this topology, synchronization messages navigate through the tree, reducing the load of the test controller (TC). Hybrid testers (HT) can aggregate synchronization messages coming from their children and send only one message to their parents. Leaf testers (LT) differ from HT, since they do not have children and do not aggregate messages. Log messages are also aggregated, since the nodes of the SUT tend to generate similar log messages. Message aggregation is ensured by specific functions that are associated to message types. For instance, when a tester receives log messages from its children, the associated function compares log messages and creates pairs containing one log message and a set of testers. Then, the set of pairs is sent to the tester's parent.

Experiments presented in a previous work [3] show that this topology has a better performance in experiments with more than 1000 nodes.

4.2. Testing approach

There are two major approaches for verifying global properties of a distributed system. The first one is to keep the output data locally in each node during the test execution and to perform a *post-mortem* analysis. This approach is less intrusive since less test data is exchanged during the execution. The second one, which we have adopted, is to perform a *live* analysis of the outputs. While this approach is more intrusive, since the exchange of test data may perturb the network performance, it is also more flexible. It allows tests to adapt themselves according to the output. This is particularly valuable when verifying liveness properties and the duration of the verification is nondeterministic.

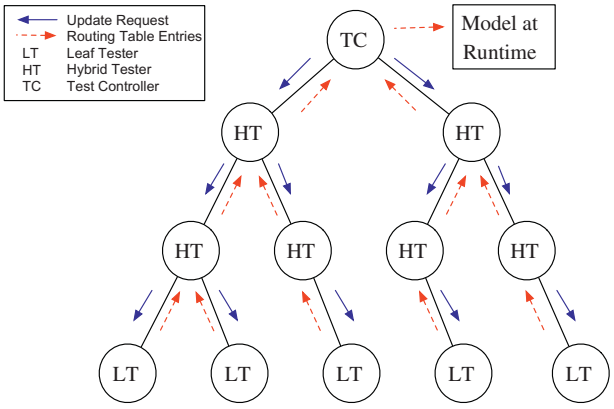

Fig. 5. Test architecture topology.

The rationale is to gather the output data on a single node, build a centralized model of the system and verify global liveness properties on this model. Once a property is verified the execution can be stopped. An important issue of this approach is finding an update frequency that is adapted to the property verification.

4.3. Global model and property specification

As stated in Section 3.2, several distributed hash tables share common properties. However, the diversity of routing and updating algorithms complicates the writing of tests to verify these properties through different implementations. The complexity can be reduced if the oracle is specified on a more abstract level, allowing tests to ignore implementation details such as the nature of identifiers, the data structure used to store the routing table, and method signatures.

Fig. 6 presents a model of the topology of distributed systems. The model is simple, yet sufficient to verify the properties presented in Section 3.2. The classes `System` and `Node` are connected by two disjoint associations, `available` (the nodes that joined the SUT) and `unavailable` (the nodes that left the SUT). Each node has a set of neighbors. The model also contains invariants that prevent nodes from been available and unavailable at the same time and from being part of its own neighborhood.

The `System` class contains two operations, **diameter ()** and **groups ()**, which calculate the diameter of a graph and the number of independent graphs, respectively. This model usually must be modified for testing a specific system or different properties. For instance, if one wants to test a load-balancing property, mentioned in Section 3.2, several new attributes (e.g., CPU load, memory usage, etc.) would be needed and the current associations would be useless.

Once the model and its invariants are done, the problem is to create and update a model instance during the execution. The

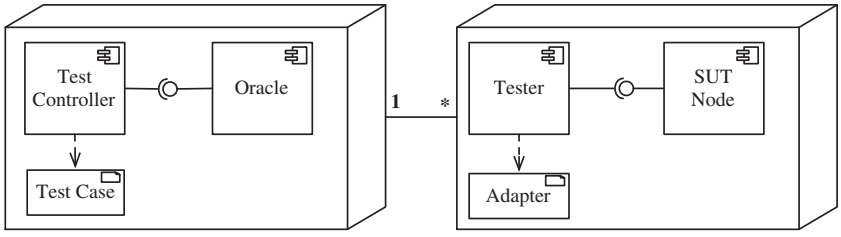

Fig. 4. Test architecture – UML deployment diagram.

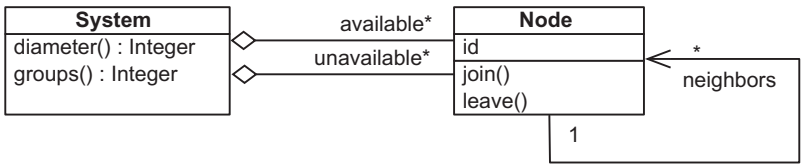

Fig. 6. UML class diagram describing system properties.

Oracle component, which is deployed along with the test controller (Fig. 4), updates and validates the global model. Thus, test sequences can directly access the global model. When a test sequence asks a tester to create a node, it also creates a new instance of `Node` and links it to the only instance of `System` through the `unavailable` association in the global model.

With this model, we can use OCL to specify the strong connectivity property in Listings 1 and 2.

The scalability property is specified in Listings 3 and 4. The first specifies an operation that returns all possible paths between two nodes and the second ensures that for each node of the system, there is a path to all other nodes in at most $\log_2(\mathcal{N})$ steps, where $\mathcal{N}$ is the size of the system.

The model also contains operations allowing the individual control of nodes. The goal is to allow the creation of test scenarios, besides the dynamic oracle. These operations must be *glued* to the SUT through adapters.

4.4. Implementation

In the current version [26], the test architecture is implemented in Java (version 1.5), global test sequences and adapters are implemented as Java methods. Meta-information about test steps (e.g., the subset of testers that should execute a step, timeout, etc.) are described as Java annotations. Listings 5 and 6 present examples of a node adapter and a test sequence, respectively.

Besides exposing nodes' interfaces, adapters also describe a method for updating the global model. When a tester receives an update request, it queries the node it controls, computes the differences with the previously sent information and sends these differences to the test controller.

We use Kermeta (version 1.4) to implement the oracle part of the test, i.e., the runtime model and the verification methods: **connectivity ()** and **diameter ()**. Since Kermeta is interoperable with Java (it compiles to Java bytecode), its integration with the test controller is flawless. The former is implemented using a depth-first search algorithm and the latter is implemented using the Floyd–Warshall algorithm [27] for the "all pairs shortest-path problem" (see its implementation in Appendix A).

4.5. Discussion

When we started the development of our experiments, we intended to use OCL to implement the oracle. Indeed, the declarative nature of OCL simplifies the specification of global properties. However, our first attempts to evaluate OCL expressions on models with several hundreds of nodes showed poor performance, which lead us to use an imperative language instead.

In our approach, we separate the oracle, test sequences, and node adapters, allowing these three parts to evolve independently.

```
context Node
def allNeighbors : Set(Node) =
    self.neighbors→union(
        self.neighbors.allNeighbors())→asSet()
```

Listing 1. All neighbors operation.

```
context System
inv:
    self.available→forAll(a,b : Node |
        a <>b implies a.allNeighbors→includes(b))
```

Listing 2. Connectivity invariant.

```
context Node
def allPaths( visited : Sequence(Node), to : Node) :
    Set(Sequence(Node)) =
    if self = to
    then
        Set{}→including(visited→including(self))
    else
        self.neighbors→collect(each : Node | node →
allPaths( visited →including(self), to))
```

Listing 3. All paths operation.

```
context System
inv:
    self.available→forAll(a,b : Node |
        a <>b implies a.allPaths(Sequence{}, b) →
exists(each : Sequence(Node)→size() ≤ self.available→size().log2()))
```

Listing 4. Diameter invariant.

Adapters depend strongly on system node interface and must be rewritten when testing different systems. Different test sequences and oracles that test the same system share the same adapters. Test sequences and oracles depend on adapters, and can be reused for testing different systems, if adapters provide the same interface. As the global model evolves, allowing the representation of new information and hence the verification of additional properties, testers must collect more information. This implies changes in the adapter, which performs additional queries on the SUT, and also in the message aggregation function. The latter is only needed if the information can be combined and reduced.

The high level of abstraction of model-based tools eases the representation and the validation of global properties. These tools ensure that the models representing the oracle data are sound (with respect to their meta-model) during and after the execution of test sequences. A possible limit of this kind of tools concerns the size of the models: most model-based tools are based on the Eclipse Modeling Framework (EMF), which is not adapted to deal with large models [28].

Our approach focus on specific classes of properties: global, liveness, observable, and controllable properties. It is not adapted to verify local properties, which do not require a global view of the system. It is not adapted to verify safety properties either, since they require an analysis of all the historical states of the system. Since we do not instrument the SUT, we cannot verify properties that are not observable from the public interfaces of the system and that do not respond to external events.

In the current implementation of the architecture, testers can force nodes to end their execution either normally or abnormally.

```
public class RoutingTableTest {
        private RemoteModel remoteModel;
        // Bootstrapper address:
        private String HOST;
        private String PORT;

        @TestStep(timeout = 40000)
        public void bootstrap() throws Exception {
            InetSocketAddress address = new InetSocketAddress(HOST, PORT);
            peer = new PastryPeer(address);
            peer.bootstrap();
            peer.createPast();
            //Store global variable with bootstrapper address:
            this.put("bootstrap", address);
        }
        @TestStep(timeout = 10000)
        public void start() throws Exception {
            //Delay to avoid bootstrap error:
            Thread.sleep(this.id() * 100);
            //Retrieve bootstrapper address:
            InetSocketAddress address = (InetSocketAddress) this.get("bootstrap");
            peer = new PastryPeer(address);
            peer.join();
            peer.createPast();
        }
        @TestStep(timeout = 2000)
        public void updateModel() throws RemoteException {
            this.push(new NodeUpdate(peer.getId(), peer.getRoutingTable());)
        }
        @TestStep(timeout = 3000)
        public void quit() throws RemoteException, NotBoundException {
            peer.leave();
        }
}
```

Listing 5. FreePastry node adapter (simplified).

```
public void testcaseExecution(TesterSet testers) {
    short tries = 0;
    int groups;
    testers.execute("bootstrap");
    testers.execute("start");

    do {
        Thread.sleep(10000);
        testers.execute("updateModel");
        tries++;
        groups = model.groups();
    } while (!connectivity() && tries < 100);

    assert connectivity();
    assert diameter() ≤ (Math.log(testers.size()) / Math.log(2));
    testers.execute("quit");
}
```

Listing 6. Test case (simplified).

This allows test sequence to inject "macro-level" faults and implement scenarios that are not interested in the origins of a failure. However, the architecture cannot inject specific faults, e.g., disk, network, bugs. We intend to combine the architecture with fault-injection tools [29] to overcome this limitation.

Another limitation of the current architecture concerns the reproducibility of tests, i.e., it does not provide repeatable automated tests [30]. In our experiments, we relied on the Grid5000 infrastructure to ensure the use of the same environment for different executions. The use of an automated staging system with support to large-scale environments (e.g., Weevil [31] and Mulini [32]) to deploy and execute tests can overcome this limitation.

5. Experimental validation

In this section, we present an experimental validation of our approach. Our objective is to validate the correct implementation and the robustness of two popular open-source DHTs with respect to the properties presented in Section 3.2: FreePastry[2] and OpenChord.[3] FreePastry is a Java implementation of the Pastry algorithm, developed by the Rice University. It has 540 classes and 89 interfaces, organized in 90 packages, for a total of 50,875 lines of code. OpenChord is an implementation of the Chord algorithm, developed by the Bamberg University. It has 96 classes and 11 interfaces, organized in 13 packages, for a total of 9245 lines of Java code.

These experiments complete our previous work [6,10], where we used these same implementations to test the functionality of their DHTs (data insertion and retrieval). These former experiments showed us that while some properties could be verified locally (at each node), some others could only be verified in a centralized manner. For our experiments, we use an incremental test methodology [6] that copes with both volatility and scalability aspects of large-scale distributed systems. The main goal of this methodology is to simplify diagnosis: tests sequences start with a small-scale system and increases the number of nodes after each execution. Node volatility is also introduced incrementally: the test sequence starts with a stable system, then with a growing system, a shrinking

[2] http://freepastry.rice.edu/FreePastry/.
[3] http://open-chord.sourceforge.net/.

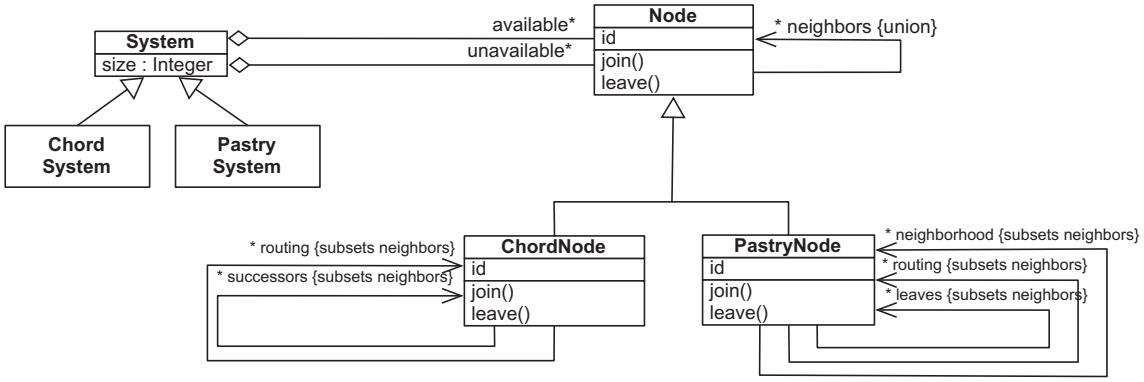

Fig. 7. UML class diagram representing chord and pastry properties.

system, and finally with a complete volatile system. We organized the experiments in the following test scenarios:

1. **Bootstrapping**: checks the ability of the SUT to build a connected (Property 1) and efficient (Property 2) system.
2. **Node isolation**: checks the ability of a node to find new neighbors, after the departure of all its neighbors (Properties 3 and 4).
3. **Expanding system**: checks the ability of nodes to update their routing tables when new nodes join the system (Properties 3 and 4).
4. **Shrinking system**: checks the ability of nodes to update their routing tables when nodes leave the system (Properties 3 and 4).

During the experiments, we used two clusters of 64 nodes running GNU/Linux.[4] In the first cluster, each node has 2 Intel Xeon 2.33 GHz dual-core processors. In the second cluster, each node has 2 AMD Opteron 248 2.2 GHz processors. Since we have full control over these clusters (nodes and network routers) during experimentation, our experiments are reproducible. The implementation and tests, produced for this paper and other P2P applications, are available on our web page.[5] We allocate the logical nodes equally through the nodes in the clusters up to 8 logical per physical node. In experiments with up to 64 logical nodes, we use only one cluster. In all experiments reported in this paper, each logical node is configured to run in its own Java VM. Execution configurations, including network, disks, DNS server, node reservation and their usage, are ensured by the OAR2 software deployed on the Grid5000 architecture.[6]

5.1. Global model extension

Fig. 7 presents an extension of the topology model introduced in Section 4.3. Here, the main superclasses `System` and `Node` have both two subclasses, which are specific to Pastry and Chord. These subclasses allow the specification of properties that only apply to these systems. For instance, we can specify that the Chord ring, built using the **successors** association, should only have one cycle. We can also specify that the Pastry nodes, connected through the **neighborhood** association are actually the physically closest nodes. This model was used to implement the test sequence presented in Section 5.

5.2. Adapter and test sequence implementation

The experiments use two adapters, one for each SUT, and three test sequences, one for each test scenario. Listing 5 presents the implementation of the FreePastry adapter. This adapter has five test steps, i.e., methods decorated with the **@TestStep** annotation. The testers use these test steps to start the bootstrap node, start a node, update the global model and quit the system. For instance, when a tester receives the message **start**, it instantiates a peer and calls successively two methods: **join ()** and **createPast ()**. The execution is bounded by a time constraint to last less than 10,000 ms, otherwise the tester aborts the execution and notifies the controller. We developed a similar adapter for OpenChord.

Algorithm 1 presents an example of a global test sequence. It specifies the test sequence presented in Section 5.3, which validates the bootstrapping process. The global test sequence creates a system with $\mathcal{N}$ nodes, waits for system stabilization and then verifies that all nodes belong to the same system (Property 1) and that the diameter of the system is $O(\log \mathcal{N})$ (Property 2). Different scenarios execute this test sequence, with $\mathcal{N}$ increasing exponentially from 16 up to 256 nodes.

The global test sequence calls two operations, defined in the global model: **diameter ()** and **groups ()**. They calculate the diameter of a graph and the number of independent graphs, respectively. The global test sequence interacts with the system nodes through the use of two messages, **start** and **bootstrap**, defined by adapters.

Algorithm 1. Global test sequence: bootstrapping test

```
Input:
S: a set of nodes;
n_b: the bootstrapper node;
limit: maximum number of checks;
delay: time between checks
begin
    tries ⟵ 0;
    send bootstrap to n_b;
    send start to 𝒩;
    wait stabilization;
    repeat
        tries ⟵ tries + 1;
        wait delay;
        log tries, groups(𝒩);
    until groups(𝒩) = 1 or tries > limit;
    log diameter(𝒩);
    assert diameter(𝒩) ≤ log_2 |𝒩|;
    assert groups(𝒩) = 1;
end
```

Listing 6 presents the Java implementation of the bootstrapping process test. For the sake of simplicity, some parts of the code were omitted. Calls to the **execute ()** method actually calls test steps on the testers side.

[4] The clusters are part of the Grid5000 platform: http://www.grid5000.fr/.
[5] Peerunit project, http://peerunit.gforge.inria.fr.
[6] http://www.grid5000.fr/mediawiki/index.php/OAR2.

Table 1
Bootstrapping test results.

Nodes	FreePastry	OpenChord
16	Pass	Pass
32	Pass	Pass
64	Pass	Pass
128	Pass	Pass
256	Fail	Fail

5.3. Bootstrapping

The first test concerns the bootstrapping process, or how a new node joins the system. In some implementations, e.g., FreePastry, OpenChord, Plan-X,[7] when a node wants to join the system, it must first contact a *bootstrap* node (i.e., bootstrapper), which will help it to get an *ID* and contact the rest of the system. The bootstrapping is a delicate process [33], especially when the whole system starts at same time. For instance, after a global outage, Skype took almost 48 h to heal in August 2007 [34].

The first test sequence creates a system with N nodes, waits for system stabilization and then verifies that all nodes belong to the same system (Property 1) and that the diameter of the system is $O(logN)$ (Property 2). Different scenarios execute this test sequence, with N increasing exponentially (2^n) from 16 up to 256 nodes.

Table 1 presents the results of the test sequence, which reveals a fault in the FreePastry bootstrapping process. Indeed, from some point between 128 and 255 nodes, the bootstrapper is unable to treat simultaneous requests. It seems that when the bootstrapper is overloaded by several parallel requests, it gives incorrect responses and induces the nodes to create several small and independent systems. This bug is particularly annoying when setting up tests for large systems, but has a workaround. The bootstrapper behaves correctly when the system is built up incrementally, respecting a delay of 100 ms between bootstrap queries.[8] The result of this test helped the developers of FreePastry to repair the bug and improve the robustness of the bootstrap process when preparing the version 2.1.[9]

The results of the test sequence also reveal a fault in OpenChord, leading to the same error found in FreePastry: the creation of several independent systems. However, in this case, the origin of the error is different. Nodes take too much time (more than 15 min) to find their neighbors and create a single system.

We use the global model to analyze the bootstrapping process, presented in Fig. 8. After initializing all nodes and requesting them to join the system, we take snapshots of the topology every 10 s. While FreePastry nodes take less than 10 s to form a strongly connected graph, OpenChord nodes are unable to do it in less than 10,000 s. After this period, there are still 4 independent systems remaining. In both systems, once Property 1 was verified, Property 2 was also verified.

5.4. Node isolation

Once we were sure that both systems respect Properties 1 and 2 (or at least 256 nodes in the case of OpenChord), we could run more elaborate tests. The second test sequence consists of two parts. First, to isolate a random node from the system. Second, to verify if such a node is able to correctly update its routing table to reach out a living node within a time limit (Properties 3 and 4). The test sequence has four steps:

[7] http://www.thomas.ambus.dk/plan-x/routing/.
[8] https://trac.freepastry.org/wiki/Planetlab.
[9] https://trac.freepastry.org/changeset/4176.

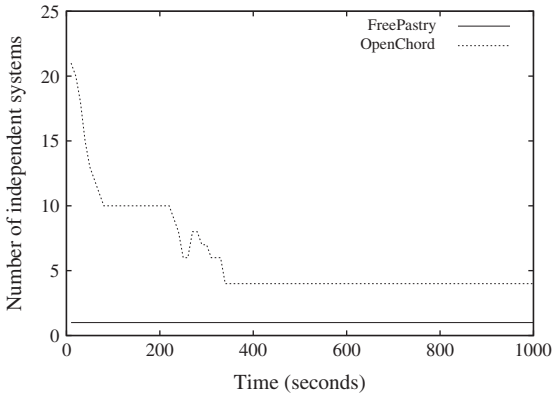

Fig. 8. Bootstrapping process (512 nodes).

1. The system is created and a set of nodes joins the system.
2. All nodes send the contents or their routing table to the global model.
3. All neighbors of a node n leave the system.
4. The routing table of n is periodically analyzed, until Property 4 (self-healing) is verified or a timeout is reached.

The routing table analysis happens as follows: the values from the updated routing table are compared with the neighbors of n before the isolation. If the intersection of these two sets of IDs is empty, then Property 4 is verified, the system is strongly connected again.

This test sequence is executed in only one test scenario, a system of 64 nodes. Indeed, creating a system with less than 64 nodes can lead the test to an inconclusive result because n may know all the nodes which are removed in the third step. In a larger system, the results should be similar since the size of the routing table would be the same.

The test showed that both implementations were able to correctly update their routing tables. OpenChord updated its routing table in about 4 s. This delay represents a unique execution of the stabilization process (whose periodicity is set to 6 s). FreePastry needed about 30 s to update its routing table and become strongly connected again. The time was bigger than OpenChord's due to the manner that the routing mechanism is updated. In the first routing attempt, FreePastry always goes through the leaf set, which is promptly updated due to its small number of entries. Proving Properties 3 and 4 is expected to be fast through the leaf set. However, in corner cases when the leaf set is not enough to answer a request (e.g., due to the number of decommissioned nodes in the isolation scenario), the other tables are used and the lazy update approach works on (i.e., only updates any address when asked).

5.5. Routing table update on an expanding system

The third test sequence checks the ability of a node to correctly update its routing table when the system is in expansion. More precisely, we verify that the nodes of a stable system take into account the new nodes that join their system. To do so, we use the global model to analyze the routing table of each node that belongs to a set of nodes N_1 to verify if it is correctly updated within a time limit, after the joining of a set of new nodes N_2.

This test case has four steps.

1. The system is started and nodes that belong to N_1 join the system.
2. Wait until the SUT reaches a stable state (Properties 1 and 2).
3. The new nodes (N_2) join the system and the global model is updated.

4. The strong connectivity of the system is verified: if all routing tables are correctly updated, then Property 4 is verified.

This test sequence is executed on different scenarios, with $|N_1| + |N_2|$ increasing exponentially (2^n) from 64 up to 1024 nodes.[10] In all executions, N_1 and N_2 have the same size. A maximum time is set to limit the test execution. This time limit starts from 8 s (allowing OpenChord to do at least one stabilization process) and increases in quadratic-logarithmic scale ($(log_n)^2$), corresponding to Property 3.

Fig. 9 shows the average time for a node to update its routing table and to get a *pass* verdict. In this scenario, FreePastry had a similar result compared with the stabilization process of OpenChord. When a new node joins a FreePastry system, it needs to communicate with all its neighbors inducing the update of their routing tables. In OpenChord, the update takes a little longer due to the time to stabilize.

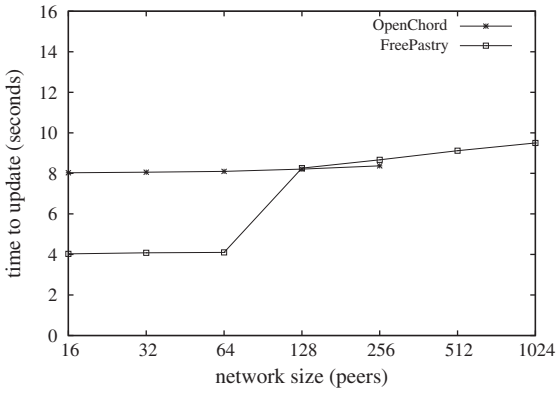

Fig. 9. Routing table update (expanding system).

5.6. Routing table update on a shrinking system

In this last test sequence, we verify that nodes that leave a stable system are correctly removed from the routing tables of the remaining nodes within a time limit. The test sequence is composed of four steps.

1. The system is created and all nodes join the system.
2. Wait until Properties 1 and 2 are verified.
3. Half of the nodes leave the system and the global model is updated.
4. Wait until strong connectivity of the system is verified again (Property 4).

In this scenario, the size of the system and the time limit also increase exponentially as described in Section 5.5. Fig. 10 shows the minimum time necessary for a node to update its routing table and get a *pass* verdict.

As expected, OpenChord shows a faster routing table updating process than FreePastry due to its stabilization process. In fact, this stabilization process showed that it can detect the departures quickly and may be a better update approach compared with FreePastry.

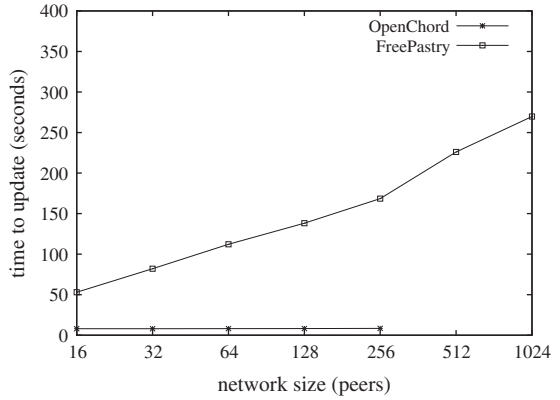

Fig. 10. Routing table update (shrinking system).

5.7. Discussion

While Properties 1, 2 and 4 may be verified using information available at the SUT interface, the verification of Property 3 is more complex. In order to measure the number of exchanged messages, one must monitor the communication on all nodes of the SUT, filter the messages that are not related to the self-organization and verify that the number of messages exchanged corresponds to $O((log\ N)^2)$. In the tests presented above, we used a different approach: we measure the time needed for self-healing in different scales. If the time increases quadratic-logarithmically, we consider that the property is respected. The tests also showed an error in the bootstrapping process of OpenChord: the time needed to create a valid system with more than 500 nodes is unsatisfactory, making the implementation unusable.

A thorough analysis of the source code, the execution log and of the resource usage revealed a design error. Indeed, when a join request arrives at the bootstrapper node, it creates one thread to process the request.[11] Thus, when several requests arrive at the same time, the node spends more time creating threads (and context switching) than actually processing the requests. Applying the Proactor design pattern [35] would fix this error.

6. Related work

In preliminary work [6,36], we presented a centralized test architecture (i.e., central coordinator managing distributed testers) for P2P systems. With this architecture, we can keep information about running tests (e.g., routing table entries) in global variables placed in the central component. Moreover, we investigated two issues: (1) we demonstrated that volatility is a key-parameter when testing a large-scale P2P system; and (2) we investigated scalability issues of such architecture resulting from the cost of coordination management of tests. During these investigations, we executed several tests on different systems. These tests were limited to oracles that used deterministic data and could be decided locally at each tester. The model-based approach presented here, allows oracles to use global information; hence, allowing us to write more elaborated tests to check global properties.

Liu et al. [37] propose an analogous approach. The authors implemented a tool, WiDS Checker, which stores events from a distributed application and uses these events to replay the execution of applications developed with WiDS. The replay uses only one simulation process. During the replay, their tool checks for predicate violations and, when a violation is caught, it can present the trace that leads to the violation. Unlike our approach, they use a pre-compiler to inject tracking code to the SUT and have access to its internal properties (white-box testing). In our approach, we do not instrument the code and only have access to public properties (black-box testing). In WiDS, verifications are done post-mortem and not at runtime.

[10] 1024 nodes correspond to 8 nodes per machine in the clusters.
[11] Classes **SocketEndpoint** and **RequestHandler**.

MaceODB [38] is to some extent, similar to WiDS Checker: it injects tracking code to a domain-specific language, Mace. The injected code is also used to check for liveness and safety properties, but at runtime. Unlike our approach, where properties are written in Kermeta, MaceODB uses a declarative language to specify properties. Thanks to this language the tool can decide which node (or set of nodes) can check for a given property and to optimize message exchange between nodes.

WiDS Checker and MaceODB are both close to our approach. However, their goal is to help debugging of distributed systems and not testing. They observe the execution of a SUT, in all its states, but do not use test sequences to drive this SUT to given states. As they do not implement a test architecture and their test code runs in the same process as the SUT, they cannot execute more complex test scenarios, involving synchronization, churn simulation, or load and scale variation.

Common test architectures for distributed systems are generally based on centralized test controller. Ulrich et al. [5,39], for instance, present a test architecture for distributed systems, called Test and Monitoring Tool (TMT), which coordinates actions by synchronization of events. This architecture uses a global tester and a distributed tester. The global tester divides the test cases in small parts called *partial test cases* (PTC). Each PTC is assigned to a distributed tester and can be executed in parallel to another PTC with respect to a function that controls mutual exclusivity. The behavior of the distributed testers is controlled by a *Test Coordination Procedure* (TCP) which coordinates the PTCs execution by synchronization events. By using synchronization events, the distributed testers do not need to control the execution of the entire test case. Each tester runs independently its partial test case. Another test architecture was presented by Walter et al. [40] for conformance, interoperability, performance and real-time testing. This architecture works as a toolbox of components (e.g., communication, test coordination, etc.) which can be combined to develop a specific test architecture. The components can be also used to simulate failures along the tests like network delay or noise. For instance, it allows creating failures in the network to simulate volatility. Because these architectures are tailored for small-scale systems and rely on a centralized topology, they scale up linearly and are not well-suited to large-scale systems.

Some test architectures are tailored for specific large-scale systems, such as: MapReduce, Publish-subscribe systems (PSS) and P2P@. The MRUnit [41] is a test framework for MapReduce systems. The framework provides a JUnit-like approach where developers write Java-based code to accelerate industry acceptance. MRUnit mocks the MapReduce system to test individually and locally the map and the reduce functions of a MapReduce job. In comparison to our approach, MRUnit exclusively addresses functional unit testing. It neither tests an entire MapReduce distributed computation, nor provides test facilities to overcome volatility issues that arise with a large number of machines.

Another large-scale testing approach is presented for PSS, that are similar to P2P systems. Some model-checking tools are based on this approach, such as Bogor [42] and Cadence SMV [43]. While they consider the volatile nature of PSS, scalability is an open issue. Along testing, nodes are simulated as threads and the size of the SUT may be bounded by the testing machine resources. Therefore, large-scale P2P systems cannot be fully tested since implementation flaws are strongly related to the system scale [6].

The Testing and Test Control Notation (TTCN-3) is a standardized test scripting language that applies to different application domains: telecommunications, protocols, aerospace, grid application workflows [44], etc. Along with the test language, TTCN-3 also provides a test execution architecture, based on the CTMF specification [45]. Similarly to our approach, TTCN-3 separates test sequences from test adapters, which are specific to the SUT. In comparison to our approach, adapters are language-independent and support data coding and decoding. The test architecture uses a centralized test controller and is not fully adapted for large-scale systems.

Other approaches also provide a language to test large-scale systems [46,47]. In fact, they differ from the TTCN-3 presenting a declarative language. These languages are analogous to the Structured Query Language (SQL) widely used to query data base systems. Our approach could take benefit from these languages to enhance testability of large-scale systems.

Network or overlay discrete-event simulation tools, such as AgentJ [48], J-Sim [49], SimJava [50] or PeerSim [51], are an interesting complementary approach to actual deployment. These tools can execute thousands or even million of nodes on a single machine, simulating network communication, and multithreading. Thus, simulation is more economic than actual deployment and can be used before actual deployment to check the adequacy of test cases [52]. Simulation can also be used to check global properties in a simpler manner, since all nodes run in the same machine. However, from a testing perspective, simulation tools have two main issues. The first issue is that they often involve rewriting a version of an application for simulation. The second is that there is no real concurrence and faults related to concurrence or load may not manifest themselves. For instance, we used PeerSim to create an OpenChord DHT with 1000 nodes and checked for bootstrap faults, similar to the ones described in Section 5.3. Contrarily to its behavior during our experiments, OpenChord could create a single system under simulation.

The Pigeon framework [53] is a network simulator, which addresses both volatility and scalability issues to test massively multiplayer online games (MMGs). The authors believe that the performance of MMGs is the most crucial problem that should be addressed since the network latency is affected by the frequent propagation of updates during a game. To monitor the network, Pigeon uses reflection and aspect-oriented programming to include additional code to the SUT. While this framework is more adapted to simulate the peculiarities of large-scale systems, it shares the same two issues of other simulation tools.

7. Conclusion

In this paper, we presented a model-based approach for checking global liveness properties that must be ensured by different large-scale distributed systems. We claim that global properties should be checked at runtime, at real scale, using non-invasive distributed testers, and that model-based testing is an expressive and adaptable technique to specify and check the global liveness properties of a system.

In our approach, test sequences put the system into states where such properties may be violated or lead to a degradation of system performance and behavior, while models provide a high abstraction level to represent a global view and the required properties of the SUT. Models are used as live oracles, which have a view of the current state of the system and can detect property violations.

Along with the approach, we presented a software test architecture for executing test cases. This architecture ensures the correct execution of test sequences, i.e., that each step of the sequence had executed correctly by all concerned node, before executing the succeeding step. In this architecture, test sequences and node adapters are written in Java, and the oracle is written in Kermeta, a dedicated language for model transformation.

We illustrated this approach by testing the reliability of two routing algorithms under churn. Results showed common flaws in both routing strategies and clear differences. For instance,

OpenChord could not build a single system (i.e., a strongly connected graph) during real-scale experiments, revealing a defect that would not have been detected without a global view of the system.

It is important to note that the approach is not limited to reliability testing. It can also target other distributed software testing techniques, such as system, load, and elasticity testing. Indeed, system testing [10] and load testing [54] were the subject of previous experiences. The approach is not limited to a specific class of distributed system either, but to specific classes of properties. More precisely, to properties that need a global view of the system to be checked, and that are only observable at specific states of the system.

In our approach, we are only interested in the state of the system at some specific points, reducing exchanged messages during a test. This choice prevents us from checking safety properties; i.e., properties that should always be true. However, these properties are typically local and could be checked using assertions. Since we chose not to instrument the code of the SUT, we can access neither the internal states of a node nor some particular attributes, such as exchanged messages raised during a given query.

In future work, going beyond the oracle issue, we will explore model-based test scenario generation. Models at runtime and search-based algorithms can help identifying and controlling the nodes in a current topology for pushing routing algorithms into their limits; for instance the isolation and rejoin of a node subset from the remaining of the system to check the connectivity property.

Appendix A. Listings

See Listing 7.

```
aspect class System {
    operation diameter() : Integer is do
var distances : Sequence<Matrix<Integer>> init Sequence<Matrix<Integer>>.new
var size : Integer init nodes.size()

from var i : Integer init 0
until i == size
loop
    distances.add(Matrix<Integer>.new.initialize(size,size))
end

distances.elementAt(0). fill (Integer.MAX_VALUE)
var index : Integer init 0

nodes.each{node |
    node.index := index
    index := index + 1}

nodes.each{node |
    node.neighbors.each{neighbor |
        distances.elementAt(0).set(node.index, neighbor.index, 1)}}

from var k : Integer init 1
until k == size
loop
    from var i : Integer init 0
    until i == size
    loop
        from var j : Integer init 0
        until j == size
        loop
            distances.elementAt(k).set(i,j, distances.elementAt(k-1).get(i,j).min(
                distances.elementAt(k-1).get(i,k) + distances.elementAt(k-1).get(k,j)))
            j := j + 1
        end
        i := i + 1
    end
    k := k + 1
end

result := 0
        from var i : Integer init 0
        until i == size
        loop
            from var j : Integer init 0
            until j == size
            loop
                result := result.max(distances.elementAt(size - 1).get(row, col))
                j := j + 1
            end
            i := i + 1
        end
    end
}
```

Listing 7. Floyd–Warshall algorithm Implementation in Kermeta.

References

[1] T. Klingberg, R. Manfredi, The Gnutella Protocol Specification v0.6, Tech. Rep., 2002. <http://dss.clip2.com/GnutellaProtocol04.pdf>.

[2] J. Dean, S. Ghemawat, Map reduce: simplified data processing on large clusters, in: OSDI, 2004, pp. 137–150.

[3] E. Almeida, J.E. Marynowski, G. Sunyé, Y. Le Traon, P. Valduriez, Efficient distributed test architectures for large-scale systems, in: ICTSS 2010: 22nd IFIP Int. Conf. on Testing Software and Systems, Natal, Brazil, November 2010.

[4] P. Joshi, H.S. Gunawi, K. Sen, Prefail: Programmable and Efficient Failure Testing Framework, University of California at Berkeley, Tech. Rep. UCB/EECS-2011-3, January 2011. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-3.html>.

[5] A. Ulrich, H. König, Architectures for testing distributed systems, in: IFIP TC6 12th IWTCS, 1999, pp. 93–108.

[6] E.C. de Almeida, G. Sunyé, Y.L. Traon, P. Valduriez, A Framework for Testing Peer-to-Peer Systems, in: 19th ISSRE, IEEE Computer Society, Redmond, Seattle, USA, 2008. 11–14 November.

[7] A. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems, in: IFIP/ACM Middleware, 2001, pp. 329–350.

[8] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, A. Solberg, Models at runtime to support dynamic adaptation, IEEE Computer, pp. 46–53, October 2009. <http://www.irisa.fr/triskell/publis/2009/Morin09f.pdf>.

[9] P.-A. Muller, F. Fleurey, J.-M. Jézéquel, Weaving executability into object-oriented meta-languages, in: L.C. Briand, C. Williams (Eds.), MoDELS, ser. Lecture Notes in Computer Science, vol. 3713, Springer, 2005, pp. 264–278.

[10] E.C. de Almeida, G. Sunyé, Y. Le Traon, P. Valduriez, Testing peer-to-peer systems, Empirical Software Eng. 15 (2010) 346–379. 10.1007/s10664-009-9124-x. <http://dx.doi.org/10.1007/s10664-009-9124-x>.

[11] L. Rabbe, CIO Update: Post-Mortem on the Skype Outage, 2010, December. <http://blogs.skype.com/en/2010/12/cio_update.html>.

[12] I. Stoica, R. Morris, D.R. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: SIGCOMM, 2001, pp. 149–160.

[13] S.C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, H. Yu, OpenDHT: a public DHT service and its uses, in: R. Guérin, R. Govindan, G. Minshall (Eds.), SIGCOMM, ACM, 2005, pp. 73–84.

[14] E. Kindler, Safety and liveness properties: a survey, Bull. Eur. Assoc. Theor. Comput. Sci. 53 (1994).

[15] J. Xu, A. Kumar, X. Yu, On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks, IEEE J. Sel. A. Commun. 22 (1) (2006) 151–163. <http://dx.doi.org/10.1109/JSAC.2003.818805>.

[16] H.I. Sitepu, C. Machbub, A.Z.R. Langi, S.H. Supangkat, Unohop: efficient distributed hash table with o(1) lookup performance, in: J.I. Agbinya, E. Biermann, Y. Hamam, N. Ntlatlapa, K. Ferguson (Eds.), BroadCom, IEEE Computer Society, 2008, pp. 76–81.

[17] P. Ganesan, P.K. Gummadi, H. Garcia-Molina, Canon in G major: designing DHTs with hierarchical structure, in: ICDCS, IEEE Computer Society, 2004, pp. 263–272.

[18] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A view of cloud computing, Commun. ACM 53 (4) (2010) 50–58. <http://doi.acm.org/10.1145/1721654.1721672>.

[19] R. Bakhshi, F. Bonnet, W. Fokkink, B.R. Haverkort, Formal analysis techniques for gossiping protocols, Oper. Syst. Rev. 41 (5) (2007) 28–36.

[20] P.-A. Muller, F. Fleurey, J.-M. Jézéquel, Weaving executability into object-oriented meta-languages, in: S.K.L. Briand (Ed.), Proceedings of MODELS/UML'2005, ser. LNCS, vol. 3713, Springer, Montego Bay, Jamaica, 2005, pp. 264–278. <http://www.irisa.fr/triskell/publis/2005/Muller05a.pdf>.

[21] J.-M. Jézéquel, Model driven design and aspect weaving, J. Software Syst. Model. (SoSyM) 7 (2) (2008) 209–218. <http://www.irisa.fr/triskell/publis/2008/Jezequel08a.pdf>.

[22] OMG, MOF 2.0 core specification, OMG, Tech. Rep. formal/06-01-01, April 2006, OMG Available Specification. <http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[23] G. Blair, N. Bencomo, R.B. France, Models@ run. time, Computer 42 (10) (2009) 22–27.

[24] A. Cruz, Update on Today's Gmail Outage, 2009, February. <http://gmailblog.blogspot.com/2009/02/update-on-todays-gmail-outage.html>.

[25] J. Grabowski, A. Wiles, C. Willcock, D. Hogrefe, On the design of the new testing language TTCN-3, in: H. Ural, R.L. Probert, G. von Bochmann (Eds.), TestCom, ser. IFIP Conference Proceedings, vol. 176, Kluwer, 2000, pp. 161–176.

[26] E.C. de Almeida, J.E. Marynowski, G. Sunyé, P. Valduriez, Peerunit: a framework for testing peer-to-peer systems, in: C. Pecheur, J. Andrews, E.D. Nitto (Eds.), ASE, ACM, 2010, pp. 169–170.

[27] R.W. Floyd, Algorithm 97: shortest path, Commun. ACM 5 (1962) 345. <http://doi.acm.org/10.1145/367766.368168>.

[28] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, J.-M. Jézéquel, An eclipse modelling framework alternative to meet the models@runtime requirements, in: R.B. France, J. Kazmeier, R. Breu, C. Atkinson (Eds.), MoDELS, ser. Lecture Notes in Computer Science, vol. 7590, Springer, 2012, pp. 87–101.

[29] M.-C. Hsueh, T.K. Tsai, R.K. Iyer, Fault injection techniques and tools, IEEE Comput. 30 (4) (1997) 75–82.

[30] R.V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[31] Y. Wang, M.J. Rutherford, A. Carzaniga, A.L. Wolf, Automating experimentation on distributed testbeds, in: D.F. Redmiles, T. Ellman, A. Zisman (Eds.), ASE, ACM, 2005, pp. 164–173.

[32] G. Jung, C. Pu, G. Swint, Mulini: an automated staging framework for QoS of distributed multi-tier applications, in: Proceedings of the 2007 Workshop on Automating Service Quality: Held at the International Conference on Automated Software Engineering (ASE), ser. WRASQ '07, ACM, New York, NY, USA, 2007, pp. 10–15. <http://doi.acm.org/10.1145/1314483.1314486>.

[33] M. Jelasity, A. Montresor, Ö. Babaoglu, The bootstrapping service, in: ICDCS Workshops, IEEE Computer Society, 2006, p. 11.

[34] V. Arak, What happened on august 16, 2007, August. <http://heartbeat.skype.com/2007/08/what_happened_on_august_16.html>.

[35] D.C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, second ed., John Wiley & Sons, Inc., New York, NY, USA, 2000.

[36] E.C. de Almeida, G. Sunyé, Y.L. Traon, P. Valduriez, Testing peers volatility, in: 23nd IEEE/ACM ASE, September 15–19, L'Aquila, Italy, 2008.

[37] X. Liu, W. Lin, A. Pan, Z. Zhang, Wids checker: combating bugs in distributed systems, Networked Syst. Des. Implementation (NSDI) (2007).

[38] D. Dao, J. Albrecht, C. Killian, A. Vahdat, Live debugging of distributed systems, Compiler Construction (2009) 94–108.

[39] A. Petrenko, A. Ulrich, Verification and testing of concurrent systems with action races, in: TestCom, August 29–September 1, Ottawa, Canada, 2000, pp. 261–280.

[40] T. Walter, I. Schieferdecker, J. Grabowski, Test architectures for distributed systems – state of the art and beyond, 1998.

[41] (2010) MRUnit Project, <http://archive.cloudera.com/docs/mrunit>. <http://archive.cloudera.com/docs/mrunit/index.html>.

[42] L. Baresi, C. Ghezzi, L. Mottola, On Accurate Automatic Verification of Publish-Subscribe Architectures, in: 29th ICSE, IEEE Computer Society, Washington, DC, USA, 2007, pp. 199–208.

[43] D. Garlan, S. Khersonsky, J.S. Kim, Model checking publish-subscribe systems, in: 10th International SPIN Workshop, Springer, 2003, pp. 166–180.

[44] T. Rings, H. Neukirchen, J. Grabowski, Testing grid application workflows using TTCN-3, in: ICST, IEEE Computer Society, 2008, pp. 210–219.

[45] I.I.S. 9646, Open systems interconnection conformance testing methodology and framework, 1991.

[46] A. Wang, P. Basu, B.T. Loo, O. Sokolsky, Declarative network verification, in: A. Gill, T. Swift (Eds.), PADL, ser. Lecture Notes in Computer Science, vol. 5418, Springer, 2009, pp. 61–75.

[47] H.S. Gunawi, T. Do, P. Joshi, J.M. Hellerstein, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, K. Sen, Towards automatically checking thousands of failures with micro-specifications, University of California at Berkeley, Tech. Rep. UCB/EECS-2010-98, 2010.

[48] I. Taylor, B. Adamson, I. Downard, J. Macker, AgentJ: Enabling java NS-2 simulations for large scale distributed multimedia applications, in: The 2nd International Conference on Distributed Frameworks for Multimedia Applications, 2006, May 2006, pp. 1 –7.

[49] J. Kacer, Discrete event simulations with J-Sim, in: J. Waldron, J.F. Power (Eds.), PPPJ/IRE, ser. ACM International Conference Proceeding Series, vol. 25, ACM, 2002, pp. 13–18.

[50] W. Kreutzer, J. Hopkins, M. van Mierlo, Simjava – a framework for modeling queueing networks in java, in: Proceedings of the 29th Conference on Winter Simulation, ser. WSC '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 483–488. <http://dx.doi.org/10.1145/268437.268548>.

[51] A. Montresor, M. Jelasity, PeerSim: a scalable P2P simulator, in: Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09), Seattle, WA, September 2009, pp. 99–100.

[52] M.J. Rutherford, A. Carzaniga, A.L. Wolf, Evaluating test suites and adequacy criteria using simulation-based models of distributed systems, IEEE Trans. Software Eng. 34 (4) (2008) 452–470.

[53] Z. Zhou, H. Wang, J. Zhou, L. Tang, K. Li, W. Zheng, M. Fang, Pigeon: a framework for testing peer-to-peer massively multiplayer online games over heterogeneous network, in: 3rd CCNC, 2006, pp. 1028–1032.

[54] J.A. Meira, E.C. de Almeida, Y.L. Traon, G. Sunyé, Peer-to-peer load testing, in: G. Antoniol, A. Bertolino, Y. Labiche (Eds.), ICST, IEEE, 2012, pp. 642–647.

All in-text references underlined in blue are linked to publications on ResearchGate, letting you access and read them immediately.